

On the Usage of Concrete Syntax in Model Transformation Rules

Thomas Baar and Jon Whittle



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Technical Report No. LGL-REPORT-2006-002
February 2006

Software Engineering Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland

On the Usage of Concrete Syntax in Model Transformation Rules

Thomas Baar¹ and Jon Whittle²

¹ École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

² Department of Information and Software Engineering
George Mason University
Fairfax VA 22030 USA
thomas.baar@epfl.ch, jwhittle@ise.gmu.edu

Abstract. Graph transformations are one of the best known approaches for defining model-to-model transformations in model-based software development. They are defined over the abstract syntax of source and target languages, described by metamodels. Since graph transformations are defined on the abstract syntax level, they can be hard to read and require an in-depth knowledge of the involved metamodels. In this paper we investigate how graph transformations can be made much more compact and easier to read by using the concrete syntax of the involved languages. We illustrate our approach by defining model refactorings.

Keywords: Metamodeling, Model Transformation, Refactoring, UML

1 Motivation

One of the key activities of model-based software development [1] is transformation between models. Model transformations are defined in order to bridge two different modeling languages (e.g., to transform UML sequence to UML communication diagrams) or to map between representations in the same language. A well-known example of the latter case is refactorings, i.e., transformations that aim at improving the structure of the source model [2, 3].

Model transformations can be expressed in many formalisms (see [4] for an overview) but graph transformation based approaches [5] are especially popular due to their expressive power. Also the recently adopted OMG standard “Query, Views, Transformations (QVT)” is based on this technique [6]. The problem tackled in this paper is that model transformations written in a pure graph transformation notation can easily become complex and hard to read (see examples given in [7] for illustration).

A transformation written in QVT consists of a set of transformation rules. Each rule has a left-hand-side (LHS) and right-hand-side (RHS) which define the patterns for the transformation rule’s source and target models. A rule is applied on a given, concrete source model by matching a sub-model of the concrete model

with the LHS of the rule and replacing the matched sub-model with the RHS, where any matchings are applied to the RHS before replacement. Additionally, all conditions imposed by the optional when-clause of the rule must be satisfied. The patterns defining the LHS and RHS are given in terms of the metamodels for the source and target modeling language (note that nowadays all major modeling languages are defined in the form of a metamodel). For the sake of simplicity in this paper (but our approach is not restricted to that), we will assume that the modeling languages for the source and target model coincide and thus each transformation rule refers only to the metamodel of one language.

A disadvantage of the above approach in defining model transformations is that the metamodel captures only the abstract syntax of the modeling language and the more readable concrete syntax is not used in the transformation rule. Transformations written purely using abstract syntax are not very readable and require the reader to be familiar with the metamodel defining the abstract syntax. These metamodels can be very complex for real languages, such as UML [8]. To overcome this problem, our approach is to write the transformation rules directly in the concrete syntax of the modeling language where possible. Unfortunately, this cannot be done directly since a number of subtleties of patterns in transformation rules have to be taken into account. In this paper, we investigate how the concrete syntax of the modeling language can be adapted for the special needs of transformation rules.

The rest of the paper is organized as follows. Section 2 gives an overview of current techniques for defining transformation rules, with an emphasis on graph transformations. We show in Section 3 how to improve the readability of transformation rules by exploiting a concrete syntax adapted from the source and target modeling language. Section 4 concludes the paper.

1.1 Related Work

The authors know of no other work in using concrete syntax for graph-based model transformations. There is a good deal of research in applying graph transformations to software engineering problems — see [9] for an introduction — such as code generation, viewpoint merging and consistency analysis. However, all of these base transformation rule definitions on abstract syntax and do not exploit the concrete syntax of the source and target modeling languages.

There are, however, approaches to *handle* the concrete syntax representation of a model if a transformation rule, which has been defined based on the abstract syntax, is applied on a given model. Guerra and de Lara [10] propose to represent the concrete syntax as part of the metamodel. [10] discusses how abstract and concrete syntax can be synchronized when applying a transformation rule. The rules themselves, however, are given in generic object diagram syntax and do not exploit the concrete syntax.

2 Defining Model Transformations

In this section, we present background on defining model transformations.

2.1 Metamodeling

A modeling language has three parts: (1) the abstract syntax that identifies the concepts of the language and their relationships, (2) the concrete syntax that defines how the concepts are represented in a human-friendly format, and (3) the semantics of the concepts. This paper is only concerned with (1) and (2).

The abstract syntax of a modeling language is usually given by a metamodel. A metamodel is given as a (simplified form of a) UML class diagram [8] with OCL [11] invariants. The concepts of the language are classes in the metamodel, with attributes, and their relationships are associations. Because these elements occur in a *metamodel*, they are called *metaclasses*, *metaattributes*, etc.

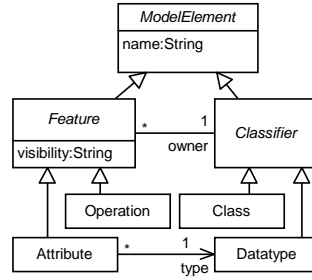


Fig. 1. Metamodel of simplified class diagrams, called *CDSimp*.

Figure 1 shows the metamodel of a drastically simplified version of UML class diagrams, called *CDSimp*. The language *CDSimp* will serve as a running example in the remainder of this paper. The metamodel for *CDSimp* consists of metaclasses that correspond directly to concrete model elements, namely **Attribute**, **Operation**, **Class** and **Datatype**, as well as abstract metaclasses that do not have a concrete syntax representation but are introduced for structuring purposes: **ModelElement**, **Feature**, **Classifier**. For instance, the metaclass **ModelElement** declares a metaattribute **name** of type **String** that is inherited by all other metaclasses.

OCL invariants attached to the metamodel impose restrictions that every well-formed model must obey (thus, the invariants are also called *well-formedness rules*). Relevant for the examples presented later in this paper are two invariants. The first invariant restricts the values for visibility and the second invariant says that the names of all features in a class or datatype are pairwise different:

```

context Feature inv:
  Set{ 'public', 'private', 'protected' } -> includes(visibility)
context Classifier inv:
  self.feature -> forAll(f1, f2 | f1.name = f2.name implies f1 = f2)
  
```

Considering only the abstract syntax of a modeling language, one can say that a *model* written in this modeling language is just an instance of the lan-

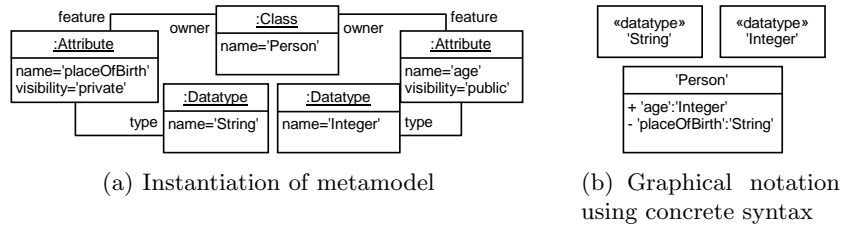


Fig. 2. Two representations of the same class diagram

guage’s metamodel and this model can be depicted as an object diagram (cf. Fig.2(a)). Instances of metamodels are not very readable, however, because all concrete model concepts are reified as metaclasses in the metamodel. More readable for humans is a graphical representation of the same model that takes the *concrete syntax* of the language into account (cf. Fig.2(b)). The concrete syntax for *CDSimp* resembles that of UML class diagrams. The only difference is that string literals, such as the name of a class or attribute, are given in quoted form (e.g. ‘Person’ instead of Person). We will need this convention later on.

2.2 Concrete Syntax Definition

The concrete syntax of a language can be defined as a mapping from all possible instances of the language’s metamodel into a representation format (in most cases a visual language [12]). It is still current practice to define the concrete syntax of a modeling language only informally. For the sake of brevity, we also give an informal definition, but, as shown in [13], it is possible without much overhead to turn an informal syntax definition into a formal one. The language *CDSimp* has the following concrete syntax definition:

- Classes and datatypes are represented by rectangles with two compartments.
- The first compartment contains the name of the class/datatype. The name of datatypes is stereotyped with `<<datatype>>`.
- The second compartment contains the representation of all owned features. A feature is represented by a line of the form:
`visiRepr ' ' name [':' type]`
 where *visiRepr* is a representation of the feature’s visibility (‘+’ for ‘public’, ‘-’ for ‘private’, ‘#’ for ‘protected’), *name* is the actual name of the feature, and, in case of an attribute, *type* is the name of the attribute’s type.

Concrete syntax definitions are needed only for those concepts that are reified in a concrete model. For example, the abstract metaclass **Feature** does not have a concrete syntax definition.

2.3 Model Transformations

The exact format and semantics of model transformations is precisely described in [6]. In this paper, we consider only the format of the patterns LHS and RHS in

each transformation rule, and the relationship of LHS and RHS to the optional when-clause of the rule.

Within a pattern, all objects are labeled by a unique variable which is declared to be of the same type as the object. The variable is shown in the first compartment as `var ':' class`. Variables are also used in order to represent concrete values in objects for attributes. Unlike usual object diagrams, objects of abstract classes (e.g. **Classifier**) can occur in patterns.

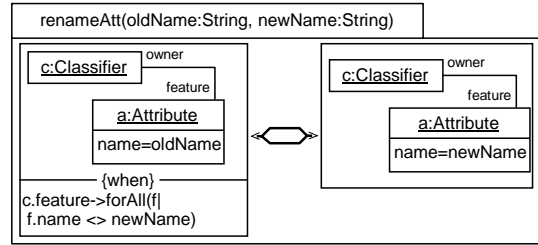


Fig. 3. QVT rule to rename an attribute within classifiers

Figure 3 shows an example for renaming an attribute of a classifier. The pattern LHS has two objects labeled with variable *c* and *a* of type **Classifier** and **Attribute**, respectively. In addition, the variable *oldName* of type **String** occurs in the slot for metaattribute **name**. The pattern RHS is identical to LHS with the exception that variable *oldName* is substituted by *newName*. Informally, the application of **renameAtt** on a concrete model would first find all classifiers (i.e. since **Classifier** is an abstract class all classes and datatypes) in the model that have an attribute with name *oldName* (matching with LHS) but no feature with name *newName* (checking the when-clause) and then change the name of attribute *oldName* in all matching classifiers to *newName*. Note that without the when-clause the transformation rule would transform some syntactically correct source models into incorrect target models where the well-formedness rule imposing unique feature names would be broken.

3 Patterns In Concrete Syntax (PICS)

Graph transformation rules, such as those given by QVT, are a very powerful mechanism to describe model transformations. Readability and scalability, however, can become a serious problem if the patterns LHS and RHS are given in object diagram syntax. The main idea of our approach is to alleviate these problems by exploiting the concrete syntax of the language whose models we want to transform. Unfortunately, we cannot apply the concrete syntax of the modeling language directly for the rendering of patterns because some important information of the pattern would be lost. We will, thus, first analyze the differences between a modeling language and the corresponding pattern language

used in transformation rules. Then, the pattern language is defined by its own metamodel, which is, as shown in Sect. 3.2, a straightforward modification of the original metamodel for the modeling language. Based on the modified metamodel, we define finally a concrete syntax for the pattern language. Since the concrete syntax defines in most cases for a pattern an elegant graphical representation, we call the pattern language PICS (patterns in concrete syntax). The term *PICS metamodel* refers to the metamodel of the pattern language, that has been derived from the metamodel of the modeling language.

3.1 Differences between models and patterns

For defining a concrete syntax for pattern diagrams the following list of differences between models (seen as instances of the modeling language’s metamodel) and patterns used in transformation rules has to be taken into account:

1. **Objects in patterns must be labeled**³ with a unique variable (e.g. the label for `c:Class` is `c`).
2. **A pattern usually represents an incomplete model** whereas object diagrams are assumed to be complete (all constraints and multiplicities of the metamodel are satisfied). For example, the patterns LHS, RHS in `renameAtt` (Fig. 3) show neither the attribute `visibility` of object `a:Attribute` nor a link to its type (an object diagram could not drop this link due to multiplicity 1 of the corresponding association end at `Datatype`).
3. **Patterns can have objects whose type is an abstract class** whereas the type of objects in object diagrams is always a non-abstract class.
4. **Patterns can contain variables to represent attribute values in objects** whereas in object diagrams such values are always literals (or ground-terms). This is a minor difference between models and patterns because literals can be easily distinguished from variable names if (i) literals of type `String` are used only in the quoted form, e.g. `'myvar'` is different from variable name `myvar`, and (ii) only such variable names are chosen that cannot be read as literals of any other type (this can be imposed by a naming convention, e.g. variable names always start with a lowercase letter).
5. **Patterns can contain negative application conditions (NACs) and multiobjects.** NACs can be seen as syntactic sugar since these conditions can be easily expressed by the `when`-clause of a pattern. Multiobjects can in most cases be expressed alternatively as well. Thus, we assume in the following that patterns contain neither multiobjects nor NACs.

3.2 Transforming the original metamodel to PICS metamodel

The important differences between models and patterns (points (1) – (3) above) can be formalized by defining a metamodel for pattern diagrams. Fortunately,

³ In many graph transformation systems including QVT the label is optional. We assume here the strict version since it will make it easier to rewrite a pattern using the concrete syntax.

this metamodel can be automatically derived from the original metamodel by applying the following changes:

- Add attribute **label:String** with standard multiplicity [1..1] to each meta-class. This change captures the mandatory labels of objects in patterns diagrams (see difference (1) in above list of differences).
- Make all attributes in the metamodel optional (by giving them the attribute multiplicity 0..1) and change all association multiplicities from x to $0..x$. Both changes reflect incompleteness of patterns (see difference (2)).
- Make all abstract classes non-abstract (see difference (3)).

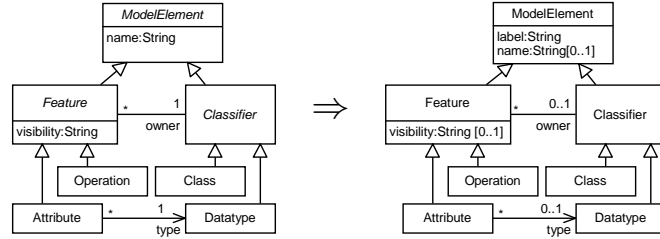


Fig. 4. Original language metamodel and derived PICS metamodel

Fig. 4 shows the changes on the metamodel for *CDSimp*. The root class **ModelElement** has a new attribute **label** that is inherited by all other classes. The two other attributes **name** and **visibility** became optional by the attribute multiplicity 0..1. The abstract classes **ModelElement**, **Feature**, **Classifier** became non-abstract and finally all multiplicities on association ends were changed to the range 0..OrigMultiplicity (note that multiplicity $*$ is not affected).

3.3 Defining concrete syntax for PICS metamodel

After the pattern language has been formalized as the PICS metamodel, we can represent each pattern as an instance of the PICS metamodel. Fig. 5(a) shows the transformation rule **renameAtt** as an example. Please note that Fig. 5(a) is just another representation of the original definition given in Fig. 3 and conveys exactly the same information. Hence, each representation equivalent to Fig. 5(a) is also equivalent to the original definition of the transformation rule.

Defining an equivalent representation for the instances of a metamodel is traditionally done by defining a concrete syntax for the metamodel. In the case of the PICS metamodel, however, the definition of a suitable concrete syntax can be challenging because: (1) the concrete syntax should be as close as possible to the concrete syntax of the modeling language whose models are being transformed; (2) the concrete syntax must handle optional occurrences of attributes and links.

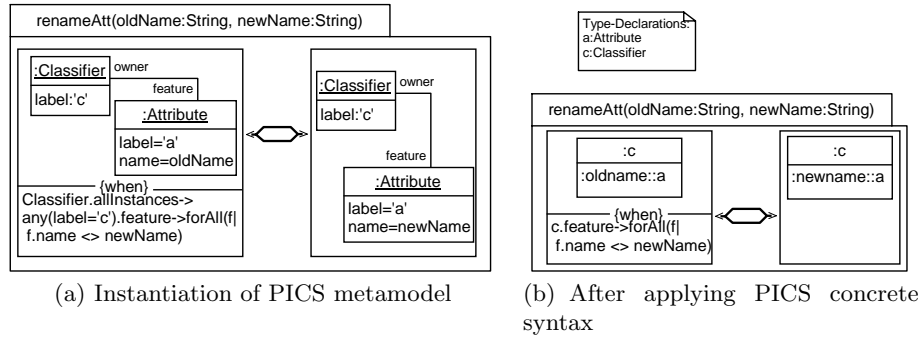


Fig. 5. Rule `renameAtt` as instance of PICS metamodel and in final notation

The first requirement is needed so that readers or writers of transformation rules need not learn an additional concrete syntax. For our running example, a concrete syntax for the PICS metamodel shown in Fig. 4 could be defined by changing the concrete syntax for CDSimp as follows:

- Instead of the *name*, the first compartment of classes/datatypes shows a line of the form *name* ':' *label* where *name* denotes the value of the optional attribute **name** and *label* the value of the mandatory attribute **label**. Since *name* appears only optionally, a delimiter ':' between *name* and *label* is needed in order to ensure correct parsing. The delimiter must not occur in *name* and *label*.
- An attribute/operation having an owning classifier is shown by a text line in the second compartment of the owning classifier. The only difference to the concrete syntax of CDSimp is the usage of delimiter ':' to separate the line items (in order to handle optional occurrences) and that the label of the attribute/operation is added at the end of the line.
In other words, the line has the form *visiRepr* ':' *name* [':' *type*] : *label*
If an attribute/operation does not have an owning classifier (note the multiplicity 0..1 for the association between **Feature** and **Classifier** in the PICS metamodel) then the text line is shown outside any other classifier.
- Instances of **Feature**/**Classifier** are rendered the same way as instances of **Operation**/**Class**.
- Instances of **ModelElement** are rendered by a one-compartment rectangle labeled with *name* ':' *label*.

The first two items explain how to adapt the renderings of classes that are non-abstract both in the original metamodel of the modeling language CDSimp and in the PICS metamodel. The rendering in PICS is very similar to that in CDSimp. Merely the label of the object had to be added and a delimiter was introduced to identify the position of an element in a text line. The last two items explain the rendering of classes that were abstract in the original metamodel but became non-abstract in PICS. Since no rendering of these classes was defined

for *CDSimp*, the new renderings for the PICS metamodel had to be invented. For some classes, e.g. **Feature** and **Classifier**, a suitable rendering can be defined as a straightforward generalization of the renderings of the subclasses. For other classes, e.g. **ModelElement**, this heuristic does not work just because the renderings of the subclasses are too diverse. An application of the PICS concrete syntax is shown in Figure 5(b) for **renameAtt**.

3.4 Optimizing the concrete syntax for PICS

Although it is always possible to define a concrete syntax for the PICS metamodel (note that showing the instance of the metamodel just as an object diagram – see Fig.5(a) for an example – would be a trivial version of a concrete syntax) it is sometimes not obvious to find an appropriate concrete syntax for PICS that is sufficiently similar to the concrete syntax of the modeling language whose models are being transformed.

Before defining the concrete syntax of PICS it is often worthwhile to check whether the transformation rules do really instantiate all classes of the PICS metamodel and really need all of the flexibility provided by the PICS metamodel. For instance, it is very likely that none of the transformation rules uses an object of class **ModelElement** since transformation rules are rarely defined on very abstract language concepts. If this is really the case then we could drop the rendering of **ModelElement** from the concrete syntax definition. In other words, the class **ModelElement** is in the *optimized PICS metamodel* an abstract class as it was already an abstract class in the metamodel of *CDSimp*. Another example is that features probably never occur in a transformation rule without their owner. Then, there is no need to invent a rendering for this case — this keeps the concrete syntax of PICS simple and similar to the concrete syntax of *CDSimp*. Again, our assumption about the form of ‘meaningful’ transformation rules could be reflected in the optimized PICS metamodel by multiplicity 1 instead of 0..1 for the association between **Feature** and **Classifier**.

4 Conclusion and Future Work

This paper addressed how to define model transformation rules in a readable and scalable way by using the concrete syntax of source and target modeling languages when defining the LHS and RHS of the rules. The concrete syntax, however, had to be adapted to the peculiarities of patterns, mainly mandatory labeling of objects and optional occurrence of attributes and links. If an intuitive concrete syntax for patterns is found, then transformation rules can be presented the same way as models of the source/target languages what takes the burden from the user to have an in-depth knowledge of the languages’ metamodels.

In practice, the definition of a suitable concrete syntax for PICS that is close enough to the syntax of the modeling languages is the main bottleneck of our approach. One problem is that the PICS concrete syntax has to invent a new rendering for classes that were abstract in the original metamodel. For all other

classes the adaptation of the existing rendering can be tricky since the PICS concrete syntax has to handle optional occurrences of attributes and links.

We propose to define the PICS concrete syntax just for those metaclasses that actually occur in transformation rules. One disadvantage of this is that rules may eventually use a metaclass for which a rendering into concrete syntax has not been defined. An alternative, for future work, is to show these metaclasses in the abstract syntax notation, that is to allow mixing concrete and abstract syntax presentations within transformation rules.

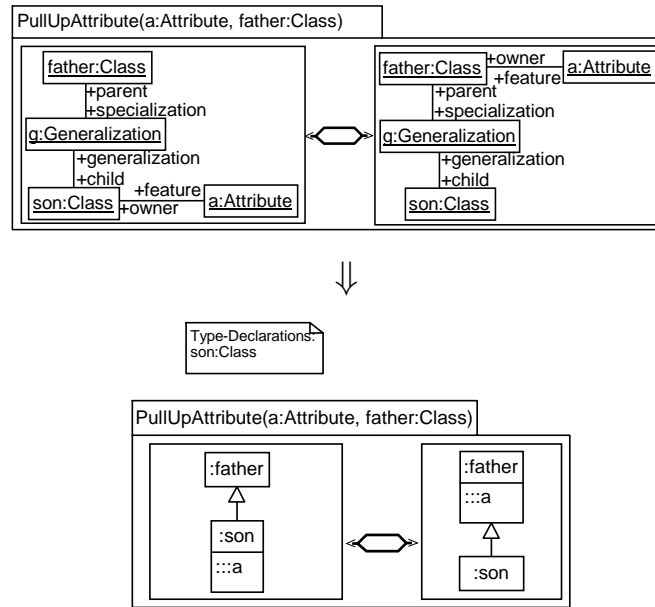
References

1. Stuart Kent. Model driven engineering. In *Proceedings of Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.
2. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.
3. Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
4. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proc. OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
5. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
6. OMG. MOF QVT Final Adopted Specification. OMG Adopted Specification ptc/05-11-01, Nov 2005.
7. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In *Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
8. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, second edition, 2005.
9. L. Baresi and R. Heckel, editors. *Tutorial introduction to graph transformation: A software engineering perspective*, 2002.
10. Esther Guerra and Juan de Lara. Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, pages 54–69, 2004.
11. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
12. Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, 13(6):573–600, 2002.
13. Frédéric Fondement and Thomas Baar. Making metamodels aware of concrete syntax. In *Proc. European Conference on Model Driven Architecture (ECMDA-FA)*, volume 3748 of *LNCS*, pages 190–204. Springer, 2005.

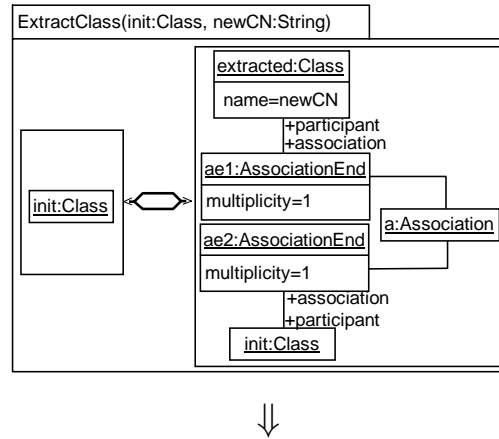
A Case Study: UML Refactoring Rules in PICS Notation

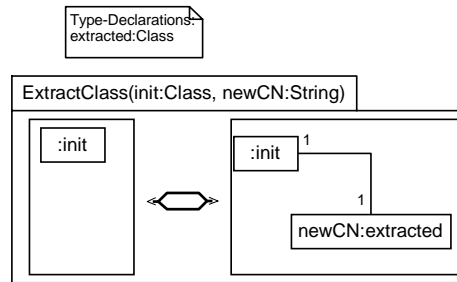
In [7], a number of refactoring rules for UML class diagrams using the abstract syntax of class diagrams has been defined. We present in the following a rewriting of these refactoring rules using our PICS approach.

A.1 PullUpAttribute



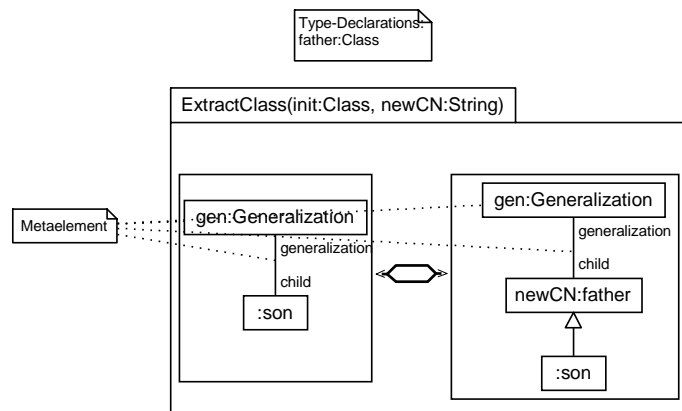
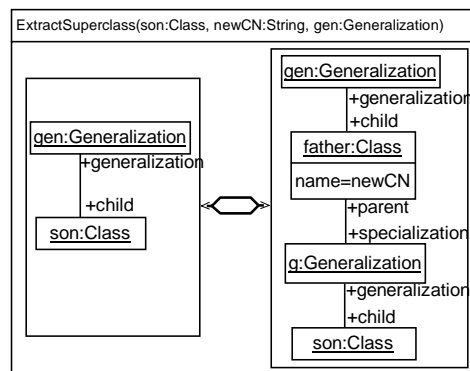
A.2 ExtractClass



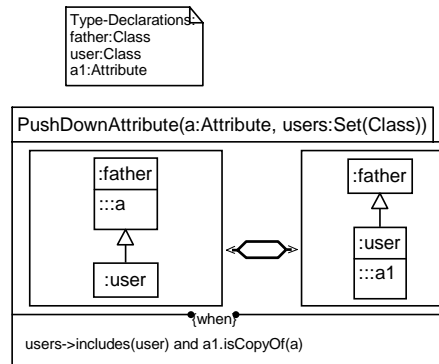
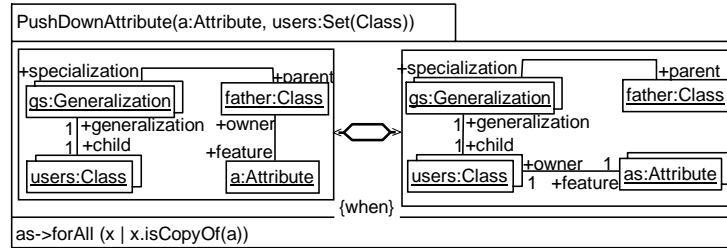


A.3 ExtractSuperClass

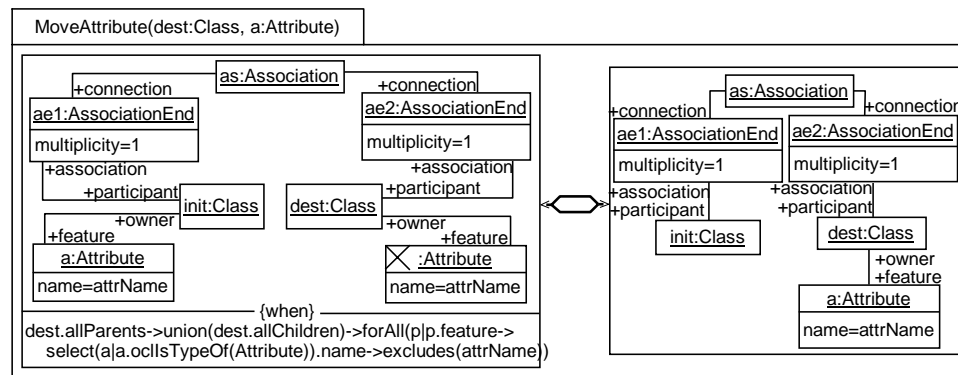
The PICS notation of `ExtractSuperClass` uses a mixture of concrete and abstract syntax (mentioned as future work in the submission).

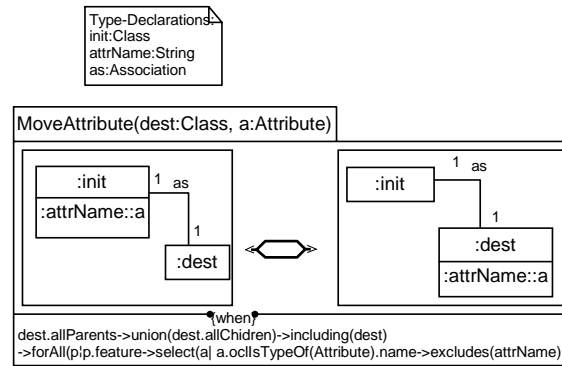


A.4 PushDownAttribute



A.5 MoveAttribute





A.6 MoveOperation

Similar to MoveAttribute.

A.7 PushDownOperation

Similar to PushDownAttribute.